NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PROOF OF FAULT COVERAGE FOR A FORMAL PROTOCOL TEST PROCEDURE

by

Michael Alan Randall

December 1992

Thesis Advisor: G. M. Lundy

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
PROOF OF FAULT COVERAGE FOR A FORMAL PROTOCOL TEST PROCEDURE

12. PERSONAL AUTHOR(S)
Michael Alan Randall

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 10/91 TO 12/92	December 1992	57

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	conformance testing, protocol specification

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
Due to the speed and complexity of communication networks being designed today, it is imperative to ensure that they operate correctly. Todays fiber optic networks, which can transmit billions of bits per second over thousands of miles, are heavily dependent on sophisticated software and protocols which are becoming increasingly difficult to test. Conformance testing is a method that is used for this purpose: to test the design of a protocol against an implementation of the design. This thesis provides some insight into the conformance testing problem by first providing background on some current protocol test methods, and then focusing on a newer method, which is based on a formal protocol specification. A proof is given that demonstrates the method's error detection capabilities. Two well known local area network protocols, Token Bus and Fiber Distributed Data Interface (FDDI), are used as examples to illustrate how the test method is applied to a specification.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: G. M. Lundy	22b. TELEPHONE (Include Area Code): (408) 646-2094; 22c. OFFICE SYMBOL: CS/Lu

Approved for public release; distribution is unlimited

Proof of Fault Coverage for a Formal Protocol Test Procedure

by

Michael Alan Randall

Naval Air Warfare Center, Aircraft Division

B.S. Computer Science, University of Maryland, Baltimore County, 1988

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

December 1992

Author:

Michael Alan Randall

Approved By:

G. M. Lundy, Thesis Advisor

Amr Zaky, Second Reader

CDR. Gary Hughes, Chairman,
Department of Computer Science

ABSTRACT

Due to the speed and complexity of communication networks being designed today, it is imperative to ensure that they operate correctly. Todays fiber optic networks, which can transmit billions of bits per second over thousands of miles, are heavily dependent on sophisticated software and protocols which are becoming increasingly difficult to test. Conformance testing is a method that is used for this purpose: to test the design of a protocol against an implementation of the design. This thesis provides some insight into the conformance testing problem by first providing background on some current protocol test methods, and then focusing on a newer method, which is based on a formal protocol specification. A proof is given that demonstrates the method's error detection capabilities. Two well known local area network protocols, Token Bus and Fiber Distributed Data Interface (FDDI), are used as examples to illustrate how the test method is applied to a specification.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

A. BACKGROUND

Protocols, in the simplest sense, are rules and procedures that control the flow of information. For centuries, long before any device that even resembles a modern day computer was ever conceived, mankind has struggled with designing precise and efficient communication protocols. In the 2nd century B.C., when communication protocols consisted of fire signals, it was observed by the Greek historian Polybius that "...it is chiefly unexpected occurrences which require instant consideration and help." Essentially, he noted that it was impossible to have a preconceived code using fire signals that could communicate these unexpected occurrences [HOLT91]. In modern day communication protocols, it is still the unexpected sequence of events that often leads to protocol failures, and the most difficult problem in protocol design is precisely that -- to expect the unexpected.

The first electronic communication protocols based on the use of the telegraph also encountered the problems associated with communicating unexpected events. However, there was almost always a human operator involved who could be relied upon to handle these problems. In current communication systems, when machines and processes rather than human operators are used, the same problems exist but now the errors can happen faster and human intervention cannot be counted on to recover from unexpected occurrences. The protocol design problem is now to determine the responsibilities of these processes and to establish procedures so that these responsibilities can be negotiated. In other words, there should be rules that govern the exchange of information, but there should also be an agreement between the communicating parties about the rules.

Designers of early networks such as the ARPAnet learned that ambiguous rules can trigger unlikely sequences of events which will ruin even the best design. Entire networks,

with thousands of attached computers can be rendered completely useless by a faulty protocol. Advances in network and telecommunication technology have resulted in increasingly complex communication protocols. Though electronic communication protocols have been around for many years, it is only recently that their complexity has begun to dramatically increase. Networks capable of transmitting billions of bits per second over thousands of miles are now in use. Consequently, the protocols being developed today are larger and more sophisticated than ever before. They try to offer more functionality and reliability, but as a result they have increased in size and in complexity. This is due to a number of factors, most notably the increased speed and capacity of current networks, but also the desire to make the most efficient use of available resources.

Because of society's critical dependence on communication networks and protocols, it is imperative to adequately test them to ensure they perform as intended. This is the goal of conformance testing.

B. OBJECTIVES

In this thesis, a conformance test method based upon a formal specification of a protocol will be investigated. This will include a review of some recent conformance test procedures, along with a discussion of the potential shortcomings which make them less than ideal for testing real-world protocol implementations. The conformance test method presented here is an improvement based upon earlier work [LUND90(b)].

The major contributions of this thesis are:

- an improved conformance test method
- a proof that demonstrates the method's error detection capabilities
- applications of the test method using real world protocols such as IEEE 802.4 (Token Bus) and ANSI X3T9.5 (FDDI).

Techniques for designing a protocol to allow for greater testability will also be discussed.

C. SCOPE

Specifically, the goal of conformance testing of communication protocols is used to verify that the behavior of a protocol implementation is consistent with its specification. Since much effort is put into formally specifying and verifying protocols, it is at least equally important to test a given implementation for functional correctness. If a formal specification of a protocol contains an error, a correct implementation of that specification should pass a conformance test only if it contains the same error. In other words, a conformance test should fail when the implementation and specification differ. The test is developed from a protocol's formal specification and is applied to an implementation of the protocol, preferably in a systematic manner. This situation is shown in Figure 1.

Figure 1 : Conceptual View of Conformance Testing

For all practical purposes we assume that the protocol implementation is essentially unknown. In other words it is simply a "black box," meaning that the test designer knows nothing about its internal workings. The only type of experiment we can do with the implementation is to provide it with sequences of inputs and observe the resulting outputs. This black box, commonly referred to as an implementation under test (IUT) in the literature, passes the conformance test only if all observed outputs match those prescribed by the formal specification [HOLT91]. It is difficult to test a protocol implementation in isolation because many protocol suites are composed of a number of independent layers. Since the layers are independent, and can be part of a number of different protocol suites, they should be tested as independent entities. Figure 2 shows the relationships of these entities.

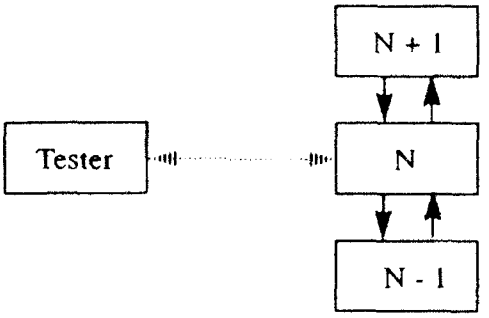

Figure 2 : Testing an Independent Layer

In Figure 2, protocol layer N is the actual implementation under test. Since, in the normal operation of this protocol, layer N must interface with upper and lower layer protocols, it needs to be tested in that context. The arrows between the protocol layers indicate the flow of data that occurs in the actual implementation.

D. ORGANIZATION

This thesis contains six chapters. Chapter II discusses some issues inherent in conformance testing, and provides four example test methods currently in use. A brief definition of the systems of communicating machines protocol model is also given. In Chapter III, the test method is presented in detail. Chapter IV is concerned with the generation of test sequences for two, well known local area network protocols: Token Bus and FDDI. This chapter also includes a discussion of the fault coverage provided by the test sequences as well as suggestions for improving the testability of the protocol specification. A proof of the fault coverage provided by this test method is given in Chapter V. This thesis is concluded with Chapter VI with discussions on the contributions of this research and ideas for further research in conformance testing.

II. CONFORMANCE TESTING

A. FUNCTIONAL TESTING

In order to understand how communication protocols can be tested, it is necessary to understand why testing is an important issue. In large, complex communication networks, administrators need a way to verify that a certain piece of equipment conforms to a prescribed standard. In an environment where thousands of phone calls or gigabits of data are being switched each second, it is not acceptable to test equipment by plugging it into the network. Ideally, we would like to verify conformance to the standard without necessarily having access to the often proprietary, internal details of the equipment.

According to [HOLT91], there are two basic goals of functional testing:

- To establish that a given implementation realizes all functions of the original specification, over the full range of parameter values.
- To establish that a given implementation can properly reject erroneous inputs in a way that is consistent with the original specification.

The trade-offs encountered in these tests are mainly between complexity and standardization. It is extremely unlikely that a test method could be devised that could test all possible behaviors of an unknown implementation by simply probing it and observing its responses. In the conformance testing of protocols, only the presence of desirable behavior can be tested. We cannot test for the presence of undesirable behavior because there is always the possibility that some untested sequence of inputs will reveal a flaw in the implementation.

B. SPECIFICATION CONFORMANCE

The process of conformance testing is based on the generation of test sequences. These test sequences attempt to exercise all parts of the protocol machine as they are defined in the specification. In the literature, the actual machine being tested is referred to as the

Implementation Under Test (IUT). Some recent conformance testing procedures involve the use of incompletely specified finite state machines with input/output labels on the transitions. The usual approach here is to represent the control portion of a protocol as a finite state machine (FSM) and generate a test sequence in the form of an input/output sequence such that, if the IUT conforms to the specification, the application of the above input sequence will generate the corresponding output sequence as specified in the test sequence. Since many protocols are not specified in the manner described above (i.e. as FSMs), an approach such as this may complicate the task of the test designer.

The common procedure followed by many current test methods is to test each edge of the FSM individually, then combine the edge sequences together to form a test sequence. Before discussing some specific test methods however, it is necessary to state some simplifying assumptions. First, we assume that the reference specification being modelled is a deterministic FSM with a known number of states. Second, the output sequence emitted by the machine occurs within a known, finite amount of time after the input sequence. Finally, every state in the FSM is reachable from every other state via one or more state transitions. For the purposes of this discussion, a *state* of an FSM (or *stop-state* as it is sometimes called) is defined as a stable condition in which the machine is awaiting an input sequence. A transition is defined as the consumption of an input signal, the possible emission of an output signal, and the possible move to a new state.

C. CURRENT TEST METHODS

The FSMs used to model a protocol specification are commonly represented by directed graphs. In a given graph, an edge (v,w) with a label a/b means that if the machine is currently in state v, then it will change to state w and output the symbol b if and only if the current input symbol is a. Many recent test methods employ this notation. Another similarity between a number of methods is the use of *Unique Input/Output sequences* (UIO sequences). UIO sequences are used within the test sequence to enhance the correctness of the test. For a given state within an FSM, a UIO sequence can be defined as a sequence of

input/output pairs which can only be observed when that input sequence is applied to that state. The purpose of such a sequence is to ensure the path by which a given state was reached. Figure 3 shows how the notation is used to label the transitions in an actual IUT. This example was taken from [YANG90] and is used because it facilitates comparisons between the different methods.

All of the states in the IUT in Figure 3 have one or more UIO sequences. For example, the sequence *a/x b/x* is a UIO sequence for state 1 because from no other state can we input *a b* and have the machine output *x x*. Similarly, a possible UIO sequence for state 5 is *c/z* because from no other state can we input *c* and have the machine output *z*. The complete list of UIO sequences for this implementation is given in Table 1. UIO sequences are important because of their uniqueness. That is, they can be used to verify their corresponding state.

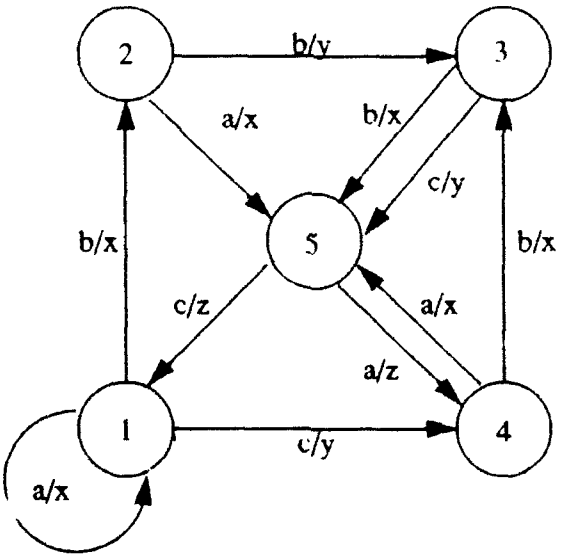

Figure 3 : Transition Diagram for an Example IUT

TABLE 1: UIO SEQUENCE FOR FIGURE 3

State	UIO sequence	Tail state
1	b/x a/x	5
1	a/x b/x	2
1	a/x c/y	4
1	a/x a/x	1
1	b/x b/y	3
1	c/y a/x	5
1	c/y b/x	3
2	b/y	3
3	b/x c/z	1
3	b/x a/x	4
3	c/y a/z	4
3	c/y c/z	5
4	b/x c/y	5
4	b/x b/x	5
5	c/z	1
5	a/z	4

As was mentioned in Chapter I, the tester cannot view the internal structure of the implementation. The only knowledge the tester has about the IUT are the outputs it emits in response to the inputs supplied. For this reason, UIO sequences are very important. They allows us to determine the state of the machine before we supplied the input sequence. The column labelled "Tail State" in Table 1 is simply the state the machine should be in after we supplied the input sequence. Determining the UIO sequences for each state is an important first step for many test procedures. The remainder of this chapter describes some recent protocol test procedures.

1. U-Method

The U-method [SABN85] was one of the first methods used in protocol testing. Essentially, it consists of 3 parts:

(1) the *RESET* transition plus the sequence from the initial machine state to the starting state of the edge being tested. (The purpose of the *RESET* transition is to "reset" the machine into its initial state, *state 1* for this machine.)

(2) the label on the edge or transition being tested.

(3) the shortest UIO sequence for the ending state (tail state) of this edge.

The complete test sequence using the U-Method is given in Table 2. A complete test sequence, or test suite, consists of the concatenation of the first, second and third parts of every test. The reader may verify that there are 77 steps in the entire test sequence.

TABLE 2: TEST SEQUENCE FOR FIGURE 3 BY THE U-METHOD

Edge Index	Edge Tested	Label Tested	Part 1	Part 2	Part 3
e1	1→1	*RESET*	*RESET*	*RESET*	*b/x a/x*
e2	1→1	*a/x*	*RESET*	*a/x*	*b/x a/x*
e3	1→2	*b/x*	*RESET*	*b/x*	*b/y*
e4	1→4	*c/y*	*RESET*	*c/y*	*b/x c/y*
e5	2→1	*RESET*	*RESET b/x*	*RESET*	*b/x a/x*
e6	2→3	*b/y*	*RESET b/x*	*b/y*	*b/x c/z*
e7	2→5	*a/x*	*RESET b/x*	*a/x*	*c/z*
e8	3→1	*RESET*	*RESET c/y b/x*	*RESET*	*b/x a/x*
e9	3→5	*b/x*	*RESET c/y b/x*	*b/x*	*c/z*
e10	3→5	*c/y*	*RESET c/y b/x*	*c/y*	*c/z*
e11	4→1	*RESET*	*RESET c/y*	*RESET*	*b/x a/x*
e12	4→3	*b/x*	*RESET c/y*	*b/x*	*b/x c/z*
e13	4→5	*a/x*	*RESET c/y*	*a/x*	*c/z*
e14	5→1	*RESET*	*RESET c/y a/x*	*RESET*	*b/x a/x*

Edge Index	Edge Tested	Label Tested	Part 1	Part 2	Part 3
e15	5→1	a/z	*RESET* a/y a/x	a/z	b/x a/x
e16	5→4	a/z	*RESET* a/y a/x	a/z	b/x a/y

The U-Method does not specify a particular order in which to test the edges; it is up to the tester to ensure that all transitions are exercised. This example begins in the initial machine state (*state 1*), and attempts to test the *RESET* transition from *state 1* to *state 1*. The first part of this test is the sequence from initial state to the starting state of the edge, in this case since we want to test a transitions emanating from the initial state, only the *RESET* transition needs to be executed. The second part of this test is the actual label on the transition, and the third part is the UIO sequence for the state in which this test ends (*state 1* in this example). The second test in the sequence is the *a/x* transition from *state 1* to *state 1*. The first, second, and third parts are *RESET*, *a/x*, and *b/x a/x*, respectively. Note that, if a state has more than one UIO sequence, we simply choose one of the shortest, and use it throughout the entire test whenever the edge being tested ends in that state.

2. RCP-Method

The RCP-method [AHO88] is similar to the U-Method in that it uses the test sequence generated by the U-Method as a starting point. It essentially concatenates the second and third parts of the U-method to form what is sometimes called a *compound edge*. A graph consisting of the compound edges is then constructed, and the Rural Chinese Postman algorithm is used to find the shortest path in the graph which traverses each edge at least once. Since this method eliminates the process of visiting the initial state after each edge sequence, (part 1 of the U-method) it results in significantly shorter test sequences. The test sequence for our example machine by the RCP-Method is given in Table 3.

TABLE 3: TEST SEQUENCE FOR FIGURE 3 BY THE RCP-METHOD

Start State	Edge Tested	Label Tested	UIO Sequence	Connection Path	End State
N/A	N/A	*RESET*	-	-	1
1	1→2	*b/x*	*b/y*	-	3
3	3→5	*c/y*	*c/z*	-	1
1	1→1	*a/x*	*b/x a/x*		5
5	5→4	*a/z*	*b/x c/y*	-	3
5	5→1	*c/z*	*b/x a/x*	-	5
5	5→1	*RESET*	*b/x a/x*	*a/z*	4
4	4→5	*a/x*	*c/z*	-	1
1	1→4	*c/y*	*b/x c/y*	*a/z b/x*	3
3	3→5	*b/x*	*c/z*	*b/x*	2
2	2→3	*b/y*	*b/x c/z*	*b/x*	2
2	2→5	*a/x*	*c/z*	*b/x*	2
2	2→1	*RESET*	*b/x a/x*	*a/z b/x*	3
3	3→1	*RESET*	*b/x a/x*	*a/z*	4
4	4→3	*b/x*	*b/x c/z*	-	1
1	1→1	*RESET*	*b/x a/x*	*a/z*	4
4	4→1	*RESET*	*b/x a/x*	*c/z*	1

The test sequence in Table 3 is constructed from the graph that is generated from the RCP tour of Figure 3. (See detail in [AHO88]). The compound edge is formed by the concatenation of the "Label Tested" and "UIO Sequence" columns. Since it is not necessary to visit the initial machine state after the test of each edge, this method simply selects one of the outgoing edges from the ending state of the previous test for the next test. For example, we start the test with the machine in an unknown state, so the *RESET* transition is necessary to bring the machine to its initial state (*state 1*). The *b/x* transition emanating from *state 1* is then tested and its UIO sequence, *b/y*, executed, putting the

machine in *state 3*. From *state 3*, the *c/y* transition is tested and its UIO sequence executed, putting the machine in *state 1*. The test sequence continues in such a manner until it arrives at a state that has no untested outgoing transitions. At this point it is necessary to execute a "connection path" transition, which takes the machine to a state which has at least one untested transition. From here, the test proceeds as before. The reader may verify that there are 55 steps in the complete test sequence, with the reduction being achieved by eliminating from the test sequence, the path from the initial state to the edge being tested.

3. MUIO-Method

In [SHEN89] it was shown that even shorter test sequences can be obtained, with the RCP-method if multiple UIO (MUIO) sequences from each state are used. The idea behind multiple UIO sequences is to reduce the repetition that results from having to follow the same UIO sequence every time an edge sequence reaches a given state. The number of times the UIO sequence for a given state will be repeated is greater than or equal to the indegree of that state. By allowing the use of different shortest UIO sequences, there is an increased possibility that we can find a UIO sequence that will end at a state with an untested outgoing transition; this effectively reduces the number of connection paths necessary and, hence, reduces the length of the sequence. The test sequence for Figure 3 by the MUIO-method is shown in Table 4.

TABLE 4: TEST SEQUENCE FOR FIGURE 3 BY THE MUIO-METHOD

Start State	Edge Tested	Label Tested	UIO Sequence	End State
N/A	N/A	RESET	-	1
1	1→1	a/x	a/x b/x	2
2	2→1	RESET	a/x b/x	2
2	2→3	b/y	b/x c/z	1
1	1→2	b/x	b/y	3
3	3→1	RESET	c/y b/x	3

Start State	Edge Tested	Label Tested	UIO Sequence	End State
3	3→5	b/x	c/z	1
1	1→1	RESET	c/y b/x	3
3	3→5	c/y	c/z	1
1	1→4	c/y	b/x c/y	5
5	5→4	a/z	b/x c/y	5
5	5→1	RESET	c/y a/x	5
5	5→1	c/z	a/x c/y	4
4	4→1	RESET	a/x b/x	2
2	2→5	a/x	a/z	4
4	4→3	b/x	b/x a/z	4
4	4→5	a/x	c/z	1

The test sequence presented in Table 4 consists of 44 steps, less than both the U-Method and RCP-Method. It is 11 steps shorter than the RCP-Method, because it eliminates the 11 connection path transitions associated with the RCP-Method. The procedure for generating this sequence is quite complicated and is not discussed here. Interested readers may consult [SHEN89].

4. MUIO-Method with Overlapping

In [YANG90] it was shown that multiple UIO sequences can be combined with overlapping to further condense the test sequence. Overlapping takes advantage of the case where some ending portion of an edge sequence is identical to the beginning portion of another edge sequence. Since the length of the test sequence is now dependent upon the length of the compound edges, one possible way to reduce the length is to combine the compound edge of one test with the next edge to be tested. Thus, when possible, these two edge sequences are combined to form one edge sequence which can be used to test both edges. By using several shortest UIO sequences, there is a good chance that an "untested"

UIO sequence can be used to verify that state. A shorter test sequence can now be obtained because the untested UIO sequence is both verifying the previous state, and beginning the test of a different transition emanating from that state. Essentially, a different incoming edge of a state is concatenated with a different shortest UIO sequence which completes the second half of the current compound edge and begins the first half of a new compound edge.

For example, the compound edge for the edge 4→5 labeled *a/x* in Figure 3 is *a/x c/z*, and the compound edge for the edge 5→1 labeled *c/z* is *c/z a/x c/y*. With overlapping, the two compound edges can be combined into the single sequence *a/x c/z a/x c/y* which can be used to test both edges. The procedure given in [YANG90] shows how to combine the maximum number of compound edges using multiple UIO sequences and overlapping. When a sequence is derived that contains the maximum number of compound edges, that sequence is often called a *fully overlapped transition sequence* (FOTS). When overlapping is combined with the MUIO method and applied to Figure 3, a minimal length test sequence of 31 steps is generated. The procedure for computing a FOTS is similar to determining sequence with the MUIO method; details can be found in [YANG90].

D. SUMMARY

The apparent goal of the U, RCP, MUIO, and MUIO with overlapping methods seems to be to shorten the length of the test sequence, and most of these techniques use complicated, optimization techniques to produce an optimal length sequence. However, since the purpose of conformance testing is to verify that the behavior of a protocol implementation is consistent with its specification, it seems that the major emphasis of the test procedure should be on the correctness of the test sequence rather than on its brevity. In other words, the purpose of a test method should be aligned with the purpose of conformance testing, which is to detect errors in faulty protocol implementations.

The techniques presented in the previous section are intended for use with a protocol described as an incompletely specified finite state machine. Since protocols are rarely

specified in this manner, the translation from specification to I/O diagram is difficult and error-prone. Protocol models which are designed for specification purposes tend to have powerful program language constructs, which simplify specification, but may prove difficult to analyze. On the other hand, models which are designed with analysis in mind, such as the CFSM model used by the previous test methods, might be considered too simple for the specification of modern, complex protocols [LUND90(b)]. What is needed is a test method that is based on a protocol specification model that eases the tasks of analysis and verification.

This paper discusses a conformance test method for generating a test sequence for a communication protocol specified as a *system of communicating machines* (SCM). The SCM specification method was designed with protocol specification and verification in mind, so it lends itself naturally to conformance testing. A number of current communication protocols such as GO-BACK-N data link protocol, Token Bus, CSMA/CD, and FDDI have been specified with SCM, and tested with the procedure presented here. Recently, a software tool was created which successfully automates the specification process using SCM [ROTH 92]. This automated tool helps the test designer to analyze the protocol specification much more easily. This does not necessarily guarantee a greater ability to produce a better test sequence, but the close relationship between the specification, verification, and test methods gives some assurance that the implementation is consistent with its specification.

E. SYSTEMS OF COMMUNICATING MACHINES

In this section the model used to specify the protocol is briefly described. A more detailed description appears in [LUND91(a)].

A *system of communicating machines* is an ordered pair $C = (M,V)$, where

$$M=\{m_1,m_2,\ldots,m_n\}$$

is a finite set of *machines*, and

$$V=\{v_1,v_2,\ldots v_n\}$$

is a finite set of *shared variables*, with two designated subsets R_i and W_i specified for each machine m_i. The subset R_i of V is called the set of *read access variables* for machine m_i, and the subset W_i the set of *write access variables* for m_i.

Each machine $m_i \in M$ is defined by a tuple $(S_i, s_0, L_i, N_i, \tau_i)$, where

(1) S_i is a finite set of states;

(2) $s_0 \in S_i$ is a designated state called the *initial state* of m_i;

(3) L_i is a finite set of *local variables*;

(4) N_i is a finite set of names, each of which is associated with a unique pair (p,a), where p is a predicate on the variables of $Li \cup Ri$ and a is an *action* on the variables of $Li \cup Ri \cup Wi$. Specifically, an action is a partial function

$$a: Li \times Ri \rightarrow Li \times Wi$$

from the values contained in the local variables and read access variables to the values of the local variables and write access variables.

(5) τ_i: $Si \times Ni \rightarrow Si$ is a transition function, which is a partial function from the states and names of m_i to the states of m_i.

Machines model the entities, which in a protocol system are processes and channels. The shared variables are the means of communication between the machines. Intuitively, R_i and W_i are the subsets of V to which m_i has read and write access, respectively. A machine is allowed to make a transition from one state to another when the predicate associated with the name for that transition is true. Upon taking the transition, the action associated with that name is executed.

The set L_i of local variables specifies a name and a range for each. The range must be a finite or countable set of values.

A *system state tuple* is a tuple of all machine states. That is, if (M,V) is a system of n communicating machines, and s_i, for $1 \le i \le n$, is the state of machine m_i, then the n-tuple $(s_1, s_2, \ldots, s_n)$ is the system state tuple of (M,V).

A *system state* is a system state tuple together with its enabled outgoing transitions. Two system states are *equivalent* if every machine is in the same state, *and* the same outgoing transitions are enabled.

The *initial system state* is the system state such that every machine is in its initial state, and the enabled outgoing transitions are the same as in the initial global state.

The *global state* of a system consists of the system state, plus the values of all variables, both local and shared. The *initial global state* is the initial system state, with the additional requirement that all variables have their initial values. A global state *corresponds* to a system state if every machine is in the same state and the same outgoing transitions are enabled.

Let $\tau(s_1,n) = s_2$ be a transition which is defined on machine m_i. Transition τ is *enabled* if the enabling predicate p, associated with name n, is true. Transition τ may be executed whenever m_i is in state s_1 and the predicate p is true (enabled). The *execution* of τ is an atomic action, in which both the state change and the action a associated with n occur simultaneously.

III. TEST METHOD

In this section, the test method is described. This description is actually a refined version of the method described in [LUND90(b)]. The input is a protocol specified as a *system of communicating machines* and the output is a complete test sequence and an I/O diagram. In order to go from the specification of a protocol machine to a test sequence, identification of the shared and local variables is necessary. The shared and local variables provide a way for the tester to provide input to and observe output from the machine during testing. The test of each edge or transition is treated as a separate, individual test, and may modify some or all of the shared and local variables.

The format of each single edge sequence test is

$$S_I\, i_1.i_2....i_n/o_1,o_2,....o_m\, S_E$$

where S_I is the state of the machine at the start of the test, $i_1,....,i_n$ are the values of the input variables prior to the test, $o_1,...,o_m$ are the values of the output variables after the test, and S_E is the state of the machine at the end of the test. The input and output variables are determined before testing begins and are taken from the shared and local variables of the machine. The procedure consists of preliminary steps, the test sequence generating procedure, and refining steps. A discussion of fault coverage follows the presentation of the test procedure. While analysis of fault coverage is not necessary to generate the test sequence, it does help determine the sequence's effectiveness.

A. PRELIMINARY STEPS

1. From the machine specification finite state diagram, mark each transition whose name appears on more than one arc. Assign to each such instance a separate distinguishing label.

2. From the predicate-action table, note the number of clauses (distinct conditions) in each enabling predicate. Mark each clause.

3. For each shared variable x, determine if x is an input variable, output variable, or both. For each x which is both, split x into two variables, x_I and x_O for testing purposes.

4. For each local variable l, determine if l is used as an interface to the higher layer user of this protocol. If so, mark l as input, output or both. Each such local variable is an input variable if it appears in an enabling predicate and an output variable if it appears in an action on the left side of an assignment arrow. If l is both input and output, split it into variables l_I and l_O for testing purposes.

Step 1 ensures that each instance of each transition is tested. A protocol specification may have the same transition name on more than one arc; we want to be certain that every arc is tested. Step 2 ensures that each clause is tested individually, if possible. An enabling predicate may consist of several clauses, any one of which might be true, allowing the transition to execute. In Steps 3 and 4, the shared and local variables are identified. Shared variables provide the means of communication between the machine and other protocol entities, and local variables allow communication with the user of the protocol. Collectively, these variables are the means the test designer has of giving inputs to the machine and observing its actions.

In some SCM specifications, additional variables are defined that are used internally by the protocol machine and are not visible to the user (upper layer(s) of the protocol) or the tester. Typically, such variables are counters or array indices. In any case, however, these variables should not appear in the test sequence since they are not under the direct control or observation of the tester.

B. TEST SEQUENCE GENERATING PROCEDURE

1. $S_t \leftarrow$ *initial_state*;

2. Let $t = (p,a)$ be an untested transition from *state*. (This notation simply means that the current transition being tested, t, has the predicate, p, as input and the action, a, as output).

(a) determine the values of the input variables which make exactly one of the untested values of p true. Check to see if these values allow any other transition from this state to be executed. If so, set additional input variables to values that ensure that only the transition being tested is enabled. Fill in the necessary input variables, and mark the others *DC* for "don't care."

(b) determine and mark the expected values for the output variables; also record the expected values assumed by the local variables.

(c) determine the expected next state and set S_E to it.

(d) determine if S_E is transient; if not, label it as a "stop state" and proceed to (3). (A state is *transient* if one or more of its enabling predicates is true upon reaching the state. This means that the machine can proceed to another state without waiting for further input from the tester).

(e) attempt to make S_E into a stop state by setting *DC* values such that, when state S_E is reached, none of the enabling predicates are true. If successful, go to (3).

(f) S_E is a transient state. If more than one transition leaving S_E is enabled, arbitrarily choose one and set inputs not yet specified, so that only one transition leaving S_E is enabled; set $t = (p,a)$ to this transition.

3. Output this test $S_I\ i_1,i_2,\ldots i_n/o_1,o_2,\ldots o_m\ S_E$ as the next test in the sequence.

4. Mark the clause just tested. If all clauses in transition t are now tested, mark t as tested. If all transitions are now marked as tested, exit to "refining steps." Otherwise, proceed to step (5).

5. Set S_I to S_E. If S_I is a stop state, proceed to (2), otherwise, proceed to 2(b).

Step 2(a) attempts to test each clause individually. This is important so that the tester knows which transition was enabled, and therefore caused the transition to occur. If it is not possible to separately test each clause, then the test designer must set the input variables so that the clauses are tested as thoroughly as possible. Perhaps they can be tested in conjunction with one another.

Steps 2(d,e,f) are concerned with *transient* states. With the existence of such states, it may be impossible to verify expected values of output variables, because the tester might not be able to determine which transition actually modified the variable. The transient state and possible multiple transition enablings that cannot be controlled in these steps, might indicate the need to modify the specification to allow for better testability.

Step 5 sets starting state of the next test in the sequence to the ending state of the current test. In order to exercise all parts of the protocol machine, some transitions may have to be executed more than once. In this instance, some judgement by the test designer may be needed. In the normal operation of the machine, if certain transitions are executed more than others, the same will likely be true during testing.

C. REFINING STEPS

1. Construct the I/O state diagram from the test sequence.

2. For each state, determine a shortest UIO sequence (if one exists). Append the UIO sequence for S_E to the end of the test sequence. If no UIO sequence for the current S_E exists, then continue testing transitions until an S_E with a UIO sequence is visited.

3. Check for converging transitions. (Converging transitions are difficult to test and may require special attention).

In Step 1, the I/O diagram is constructed from the test sequence and is a tool to help the test designer ensure completeness. This diagram is constructed directly from the test sequence with the knowledge of "stop states." The directed arcs in the I/O diagram are labeled with the corresponding values of the input and output variables. Transient states will not appear in the I/O diagram.

The purpose of the final UIO sequence in Step 2 is to verify that the last state which was reached in the test sequence is indeed the current state of the protocol machine. Since the details concerning the implementation of the machine are assumed to be "hidden" from the tester, the existence of at least one state with a UIO sequence is necessary. Without the

UIO sequence, there is no way of knowing that the last transition tested left the machine in the expected state.

Converging transitions, identified in Step 3, are distinct transitions, with identical labels, which originate from different states but end in the same state. They may arise naturally in the specification of a protocol, but should be marked for careful observation. The existence of such transitions complicates the role of the test designer and makes error detection difficult. The example protocol machine tested in a Chapter IV contains converging transitions, and the analysis of the test sequence generated from this machine will reveal their effects on the quality of the test sequence.

D. FAULT COVERAGE

For the fault coverage for the protocols presented in this paper, four classes of faults are considered [MILL90].

- Type 1: The output of an edge is altered.
- Type 2: The input of an edge is altered (without causing non-determinism).
- Type 3: A tail state of an edge is altered.
- Type 4: A head state of an edge is altered (without causing non-determinism).

Head state and tail state refer to the state in which an edge begins and ends, respectively, and a fault can be defined as a transition in the implementation under test which is not defined in the specification. When testing for faults in an implementation of a protocol, detecting faults of types 1, 2 and 4 are generally trivial. It turns out that a test for the presence of those categories of faults is automatically performed when a test sequence is checked for the presence of a type 3 fault.

For example, a type 1 fault would be detected when the test of an sequence produced output different from that which was expected. A type 2 fault would be discovered when, for instance, no output was generated in response to an input which should have caused a change in an output variable. In this case, the machine wouldn't execute the transition because the enabling predicate(s) would not be enabled. A type 4 error is typically detected in a similar manner. The non-determinism requirement for type 2 and type 4 errors is

necessary if, in the IUT, the enabling predicate of one edge is altered such that it is equivalent to another enabling predicate emanating from that edge. This would result in a non-deterministic protocol machine, which is clearly untestable.

Detection of type 3 faults are the most difficult to detect since a single error can go undetected until the very last edge sequence test. Moreover, if the error occurs in an edge that exists more than once in the specification, the fault might not be detected. It is worth noting that, when an error is detected by a test sequence, the tester cannot deduce what type of error exists in the protocol implementation; only the existence of some type of error is known.

IV. APPLICATIONS OF TEST METHOD

In this section the test generating procedure is illustrated using two well-known local area network protocols: Token Bus and FDDI. The protocols are first specified as a *system of communicating machines* and then the procedure is given.

A. TOKEN BUS PROTOCOL SPECIFICATION

The specification of the Token Bus network, originally presented in [LUND90(a)], consists of the predicate action table (Table 5), the specification for each machine, given in Figure 4, and the shared variable *MEDIUM*, also shown in Figure 4. This single shared variable is used to model the bus, which is "shared" by each machine. A transmission onto the bus is modelled by a write into the shared variable. The fields of this variable correspond to the parts of the transmitted message. The first field, *MEDIUM.t*, takes the values T, or D, which indicate whether the frame is a token or data frame; the second field, DA, contains the address of the station to which the message is transmitted (DA for "destination address"); the next field, SA, contains the originator's address (SA for "source address"); finally, the *data* field contains the data block itself.

The network stations, or machines, are defined by a finite state machine, a set of local variables, and a predicate-action table. The *initial state* of each machine is state 0, and the shared variable is initially set to contain the token with the address of one of the stations in the "DA" field.

The value of local variable *next* is the address of the next downstream neighbor, and this is initialized so that the entire network forms a logical ring. Local variable *i* is used to store the station's own address, and as implied by their names, local variables *outbuf* and *inbuf* are used for storing data blocks to be transmitted to, or received from other machines on the network. *Outbuf* is an array, and can store a potentially large number of data blocks. The local variable *j* is used to index into this array. The variable *ctr* serves to count the

number of blocks sent; it is an upper bound on the number of blocks which can be sent during a single token holding period.

The initial state of each machine is state 0, local variables *j* and *ctr* are initially set to 1, and *inbuf* and *outbuf* are set to empty. The shared variable MEDIUM initially contains the token, with the address of one station in the DA field. Thus the initial system state tuple is (0,0,..,0) and the first transition taken will be *get-tk*, executed by the station which has its local variable *i* equal to *MEDIUM.DA*.

Each machine has four states. In the initial state, 0, the station is quiescent, merely waiting to either receive the token, or a message from another station. If the token appears in the variable *MEDIUM* with the station's own address, the transition to state 2 is taken. When taking the *get-tk* transition, the machine clears the communication medium and sets the message counter, *ctr*, to 1. In state 2, the station transmits any data blocks it has, moving to state 3, or passes the token, returning to state 0. In state 3, the station will return to state 2 if any additional blocks are to be sent, until the maximum count *k* is reached. When the count is reached, or when all the station's messages have been sent, the station returns to state 0.

The receiving station, as with all stations not in possession of the token, will be in state 0. The message will appear in MEDIUM, with the receiving station's address in the DA field. The *rcv* transition to state 1 will then be taken, the data block copied, and the MEDIUM cleared by the *ready* transition. By clearing the medium, the receiving station enables the sending station to return to its initial state (0) or to its sending state (2).

For this simplified specification, the channel is assumed to be error free. This means that the clearing of the medium by the receiver may be taken as an acknowledgment by the sender. Thus, there is no need for timers, time-outs or error checking on the channel. Although many of the finer details of the IEEE Token Bus network are omitted, this specification contains the main idea of the Token Bus protocol, and provides enough detail for the generation of a test sequence.

TABLE 5: PREDICATE ACTION TABLE FOR THE NETWORK NODES

transition	predicate	action
rcv	MEDIUM.(t,DA) = (D,i)	$inbuf \leftarrow MEDIUM$ (SA, data)
ready	true	$MEDIUM \leftarrow \varnothing$
get-tk	MEDIUM.(t,DA) = (T,i)	$MEDIUM \leftarrow \varnothing$; $ctr \leftarrow 1$
pass	$outbuf[j] = \varnothing$	$MEDIUM \leftarrow (T, next, i, \varnothing)$
Xmit	$outbuf[j] \neq \varnothing$	$MEDIUM \leftarrow outbuf[j]$: $ctr \leftarrow ctr \oplus 1$; $outbuf[j] \leftarrow \varnothing$; $j \leftarrow j \oplus 1$
moreD	$MEDIUM = \varnothing \wedge$ (outbuf[j] $\neq \varnothing$ $\wedge$ ctr $\leq$ k)	- - -
pass-tk	$MEDIUM = \varnothing \wedge$ (outbuf[j] = $\varnothing$ $\vee$ ctr = k + 1)	$MEDIUM \leftarrow (T, next, i, \varnothing)$

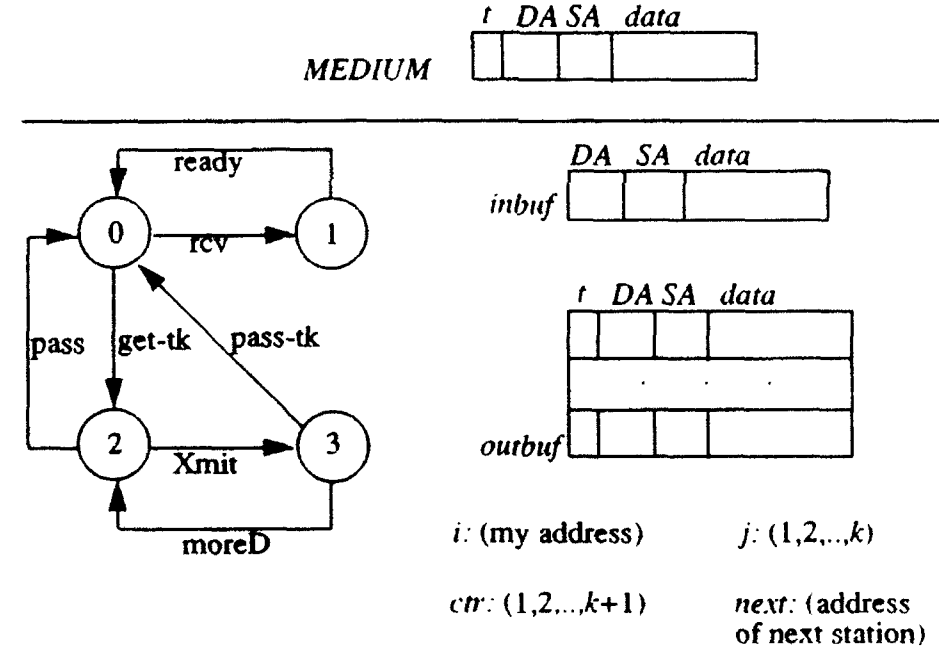

Figure 4 : Specification of Network Nodes

B. TEST SEQUENCE GENERATION

First the preliminary steps are carried out; these determine the exact format of the tests (i.e., the input and output variables). Then the tests are generated. The numbering below corresponds to the steps in the test procedure, for ease of reference.

1. Preliminaries

(1) All transition labels are unique, so no action is required.

(2) The *pass-tk* transition has two distinct clauses: $(MEDIUM = 0 \wedge outbuf[j] = 0)$ and $(MEDIUM = 0 \wedge ctr = k+1)$. We will mark the former as *pass-tk'* and the latter as *pass-tk*. (Thus, the *pass-tk'* transition represents the case when the machine has no more data to send during the current token holding period and the *pass-tk* transition represents the case when the machine has sent the maximum number of messages during the current token holding period).

(3) The shared variable *MEDIUM* is both an input and an output variable.

(4) The local variable *outbuf* is both an input variable and an output variable, and *inbuf* is an output variable only.

Note that in Step 2, the *more-D* transition is not separated into two different clauses because all three conditions must be true for the transition to be enabled. Also note that *i*, *next*, *j*, and *ctr* are local variables, but since they are not used as an interface to a higher layer user of this protocol, they are not "visible" to the tester and are not shown in the test sequence.

From these preliminary steps, we can see that the test will take the following form:

$$S_I\ MEDIUM_I\ outbuf_I\ /\ MEDIUM_O\ outbuf_O\ inbuf\ S_E.$$

Now we are ready to begin generating the test sequence.

2. Sequence Generation

(1) We begin in the initial state, 0. In step 2, we may choose any untested transition emanating from state 0; take the *get-tk* transition.

2(a) According to the predicate-action table, to enable this transition the t field of *MEDIUM* must be set to *T* and the DA field to the station's address, which is assumed to be *i*. The remaining fields may be any values, and are indicated by an '*x*' in Table 6. The other input variables are set to "don't care" or *DC*.

2(b) When the transition occurs, MEDIUM is set to empty.

2(c) S_E is set to the expected end state for this test, which is state 2.

(3) Noting that the next state is a stop state, this completes the first test in the sequence, and the appropriate values are output (Table 6).

(4) This clause and transition are now marked "tested."

(5) The value of S_t is now set to 2, and another iteration starting at step 2 is called for.

TABLE 6: TEST SEQUENCE FOR THE TOKEN BUS PROTOCOL

transition	S_I	$MEDIUM_I$	$outbuf_I$ (1 2)	$MEDIUM_O$	$outbuf_O$ (1 2)	inbuf	S_E
get-tk	0	(T,i,x,x)	DC DC	∅	- -	-	2
Xmit	2	DC	X DC	X	∅ -	-	3
moreD	3	∅	DC Y	-	- -	-	2
Xmit	2	DC	DC Y	Y	- ∅	-	3
pass-tk	3	∅	X DC	(T,next,i,∅)	- -	-	0
rcv	0	(D,i,x,x)	DC DC	-	- -	(x,x,SA, data)	1
ready	1	DC	DC DC	∅	- -	-	0
get-tk	0	(T,i,x,x)	DC DC	∅	- -	-	2
pass	2	DC	∅ DC	(T,next,i,∅)	- -	-	0
get-tk	0	(T,i,x,x)	DC DC	∅	- -	-	2
Xmit	2	DC	X DC	X	∅ -	-	3
pass-tk'	3	∅	DC ∅	(T,next,i,∅)	- -	-	0

The next iteration of the procedure arbitrarily selects the *Xmit* transition, and the values selected are shown as the second test entered in Table 6. The expected ending state of this second test is 3.

At the next iteration, the *moreD* transition is chosen, followed by the *Xmit* transition back to state 3. We now exercise the *pass-tk* transition using the $(MEDIUM = 0 \wedge ctr = k+1)$ predicate, which leads us back to the initial state of 0.

The remaining untested transitions are executed in a similar manner resulting in a final test sequence of 12 tests. The values of the input and output variables for all of these tests are shown in Table 6.

3. Fault Coverage for the Token Bus Test Sequence

The procedure for determining fault coverage consists of taking each outgoing transition from each state in the specification and modifying that transition so that it has a different tail state. For example, in a correct implementation of the Token Bus protocol, the *get-tk* transition has a tail state of 2. In an incorrect implementation, the *get-tk* transition could have a tail state of 0, 1, or 3. In this specification, Figure 4, there are 4 states and 7 transitions, so there are 28 possible tail states. Subtracting the 7 tail states in a correct IUT, leaves 21 possible type 3 errors. These are listed in Table 7.

TABLE 7: POSSIBLE TYPE 3 ERRORS FOR FIGURE 4

State	Transition	Possible End State	Result
0	*get-tk*	0	detected
0	*get-tk*	1	detected
0	*get-tk*	3	detected
0	*rcv*	0	detected
0	*rcv*	2	detected
0	*rcv*	3	detected

State	Transition	PossibleEnd State	Result
1	*ready*	1	detected
1	*ready*	2	detected
1	*ready*	3	detected
2	*Xmit*	2	detected
2	*Xmit*	0	detected
2	*Xmit*	1	detected
2	*pass*	2	detected
2	*pass*	1	detected
2	*pass*	3	**undetected**
3	*moreD*	3	detected
3	*moreD*	0	detected
3	*moreD*	1	detected
3	*pass-tk*	3	detected
3	*pass-tk*	1	detected
3	*pass-tk*	2	**undetected**

As an example of how this test sequence can be used to detect an error, consider the error that could be associated with the *rcv* transition. In order for the rcv transition to be executed, the machine must be in *state 0* and the predicate *MEDIUM.(t,DA) = (D,i)* must be true. The *rcv* transition then places the source address of the sender and the data into local variable *inbuf* (*inbuf* ← *MEDIUM.(SA,data)*). If this transition were to end in *state 0*, then either the *rcv* transition must be executed again, or the *get-tk* transition is executed. If the *rcv* transition is executed repeatedly, the machine will be in deadlock, and the error detected. If, on the other hand, the *get-tk* transition is executed, *MEDIUM* will be modified

(*MEDIUM* ← ∅) mistakenly, and the error will be detected. This type of procedure is applied to every possible, single type 3 error to determine if it can be detected.

A check of the 21 possible errors against the test sequence (Table 7) shows that all faults will be detected with the following 2 exceptions: (1) when the *pass* transition ends in state 3 (instead of state 0) and (2) when the *pass-tk* transition ends in state 2 (instead of state 0). Closer inspection reveals that these particular errors are not detected because they involve converging transitions.

Note from Table 4 that the enabling predicates and the corresponding actions for the *pass-tk* and *pass* transitions are almost identical. Furthermore, both transitions emanate from different states but end in the same state, *state 0*. If, in a faulty implementation, transition *pass-tk* ended at *state 2*, the error would go undetected because the machine would instantaneously move from *state 2* to *state 0*; this is the correct state in an implementation where the error in the *pass-tk* transition does not occur. The *pass* transition is executed because the enabling predicate for the *pass* transition must be "true" in order for the *pass-tk* transition to have executed earlier. In general, errors involving converging transitions are difficult to detect and are given further treatment in chapter V.

C. FDDI PROTOCOL SPECIFICATION

The Fiber Distributed Data Interface (FDDI) is a high speed, token ring, local area network recently developed by the American National Standard Institute (ANSI). Because of its reliability and high speed, FDDI is a very complex protocol. This is evidenced by its specification. The ANSI specification of this protocol contains two distinct machines, each with six states, and a total of approximately 60 transitions. Furthermore this work contains some undesirable ambiguities that could complicate testing. Recent work [LUND91(b)] involving FDDI has produced an *SCM* specification of an improved FDDI protocol which allows for proofs of protocol correctness and for the development of test procedures. This improved specification, though it is more precise than the ANSI specification, is also more complex; it includes two machines with a total of more than 100 states and 250 transitions.

In order to effectively test such a specification, it is necessary to break each machine into distinct, separately testable entities, and perform localized testing; that is the approach taken here. The specification provided here is a portion of the MAC receiver state diagram illustrated in [LUND91(b)]. Since this work involved an improved FDDI specification, some refinements to the example machine were necessary to make it more closely aligned with the ANSI specification. The purpose of the machine used in this example is, essentially, to analyze the stream of symbols being received by the MAC layer and determine whether or not they represent the token.

This specification consists of the predicate action table (Table 8) and the state diagram of the receive token machine (Figure 6). A complete description of the notation used in this example is beyond the scope of this thesis, however interested readers can consult [LUND91(b)]. The brief description that follows should be sufficient to allow for an understanding of the operation of the example machine. For the predicates in Table 8, each symbol represents a field of an FDDI frame; this frame format is shown in Figure 5.

Figure 5 : Frame Format

Each field of the frame has the following meaning:

- Preamble (PA) - consists of 16 or more idle symbols ($I_1..I_{max}$) that signal a start transition for synchronization of station's clock
- Starting Delimiter (SD) - consists of two symbols (J and K) that signal the start of receiving a frame
- Frame Control (FC) - consists of two data symbols. For our purposes, FC will either contain (n,n) indicating a bit pattern other than the token, or FC will be equal to Token, which indicates that the frame contains a the token.

- Destination Address (DA) - consists of a number of symbols that indicate the destination address of the frame.

- Source Address (SA) - consists of a number of symbols that indicate the source address of the frame.

- Information (INFO) - consists of zero or more symbols that represent the message carried by the frame.

- Frame Check Sequence (FCS) - consists of a number of symbols that serve to detect errors within the frame.

- Ending Delimiter (ED) - consists of one terminate symbol (T) that indicates a frame ending. The field is necessary to provide a criteria for acceptance of a valid frame. The ED must be met before a frame is accepted.

- Frame Status (FS) - consists of control indicator symbols that follow the ending delimiter of a frame.

The example machine has four states. In the initial state, 0, the machine is quiescent, merely waiting to either process a symbol stream or to reset. If *MAC_Reset* is true, a reset occurs and the machine remains in its initial state. If the preamble (PA_i) is equal to I_i, then the first idle symbol has been detected in the symbol stream, and the machine moves to *state 1*. From *state 1*, a reset or invalid signal from the physical layer would cause the machine to return to its initial state; however, if PA_i is equal to the maximum number of idle symbols and the first symbol of the starting delimiter (J) has been received, the machine moves to *state 2*. The machine would then move to *state 3* if the rest of the starting delimiter was received correctly and the frame control symbol was set to the token. From *state 3*, the *strip_on_tk* transition would be taken if a preamble is detected with only one idle symbol, and the frame control indicates a value other than the token. The *format_error_on_tk* would be exercised if, in addition to the above conditions, the ending delimiter was not equal to the terminate symbol or the preamble did not contain an idle

symbol. Finally, the *token_rcvd* transition is taken if the terminate symbol is received correctly.

It is important to emphasize that this specification comprises a very small portion of the entire FDDI specification, and the only way to test such a complex specification is to break it down into more manageable parts. The example given here is representative of the complexity that would be associated with other machines in the FDDI specification.

TABLE 8: PREDICATE ACTION TABLE FOR RECEIVE TOKEN MACHINE

transition	predicate	action
Reset	MAC_Reset = true	T_Neg ← T_Max;
Signal_Start	PH_Indication(symbol) = PAr[I_1] ∧ (symbol = $PA_r[I_1]$)	TVX ← reset; TVX ← enabled; SIGNAL idle;
Invalid	PH_Invalid = true	SIGNAL FO_Error;
Start	PH_Indication(symbol) = $PA_r[I_1..I_{max}]$ $SD_r[J]$	Idle ← off; SIGNAL RC_Start;
Token	PH_Indication(symbol) = {$PA_r[I_1..I_{max}]$, $SD_r[JK]$} ∧ FC_r = Token	SIGNAL PDU_Tk;
Strip_on_Tk	PH_Indication(symbol) = {$PA_r[I_1..I_{max}]$, $SD_r[JK]$, $FC_r[n,n]$, $PA_r[I_1]$}	SIGNAL idle;
Format_Error _on_Tk	PH_Indication(symbol) = $PA_r[I_1..I_{max}]$, $SD_r[JK]$, $FC_r[n,n]$, (¬ $PA_r[I_1]$ ∨ ¬$ED_r[T,T]$)	SIGNAL FO_Error;
Token_Rcvd	PH_Indication(symbol) = $PA_r[I_1..I_{max}]$, $SD_r[JK]$, $FC_r[n,n]$, $ED_r[T,T]$	SIGNAL Tk_Rcvd;

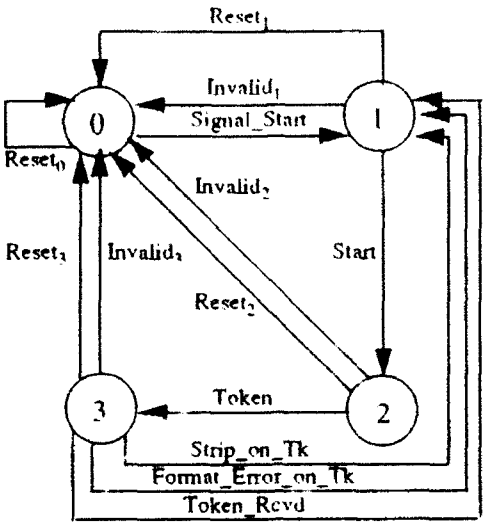

Figure 6 : FDDI Receive Token Specification

D. TEST SEQUENCE GENERATION

These steps are similar to those provided for the Token Bus test sequence. Again, the numbering below corresponds to the steps in the test procedure.

1. Preliminaries

(1) There are four reset transitions and three invalid transitions, and these are labeled with subscripts corresponding to the number of the state from which they originate.

(2) Each enabling predicate has one clause.

(3) The shared variables *T_Neg* and *TVX* are output variables.

(4) The local variables *PH_Indication*, *PH_Invalid*, and *MAC_Reset* are input variables and local variables *Idle*, *RC_Start*, *PDU_Tk*, *FO_Error*, *TK_Rcvd*, and *Flags* are output variables.

From these preliminary steps, we can see that the test will take the following form:

S_I *PH_Indiation PH_Invalid MAC_Reset* / *T_Neg TVX Idle RC_Start PDU_Tk FO_Error TK_Rcvd Flags* S_E

Now we are ready to begin generating the test sequence.

2. Sequence Generation

(1) We begin at the initial state, 0. In step 2, we may choose any untested transition emanating from state 0; take the $reset_0$ transition.

2(a) According to the predicate action table, to enable this transition, *MAC_Reset* must be set to "true." The remaining input variables are set to DC.

2(b) When the transition occurs, *T_Neg* is set to *T_Max*.

2(c) S_E is set to the expected end state for this test, which is *state 0*.

(3) Noting that the next state is a stop state, this completes the first test in the sequence. The appropriate values are now output (Table 9).

(4) This transition is now marked "tested."

(5) The value of S_I is now set to 0, and another iteration starting at step 2 is called for.

TABLE 9: FDDI RECEIVE TOKEN TEST SEQUENCE

transition	S_I	PH_Indication (symbol)	PH_Invalid	MAC_Reset	T_Neg	TVX	Idle	Rc_Start	PDU_Tk	FO_Error	TK_Rcvd	Flags	S_F
$Reset_0$	0	DC	DC	true	T_Max	-	-	-	-	-	-	-	0
Signal_Start	0	$PA_r[I_1] \wedge$ symbol $= PA_r[I_1]$	DC	DC	-	reset enable	on	-	-	-	-	-	1
$Invalid_1$	1	DC	true	DC	-	-	-	-	-	on	-	-	0
Signal_Start	0	$PA_r[I_1] \wedge$ symbol $= PA_r[I_1]$	DC	DC	-	reset enable	on	-	-	-	-	-	1
$Reset_1$	1	DC	DC	true	T_Max	-	-	-	-	-	-	-	0
Signal_Start	0	$PA_r[I_1] \wedge$ symbol $= PA_r[I_1]$	DC	DC	-	reset enable	on	-	-	-	-	-	1
Start	1	$PA_r[I_1..I_{max}]$, $SD_r[J]$	DC	DC	-	-	off	on	-	-	-	clear	2
$Reset_2$	2	DC	DC	true	T_Max	-	-	-	-	-	-	-	0
Signal_Start	0	$PA_r[I_1] \wedge$ symbol $= PA_r[I_1]$	DC	DC	-	reset enable	on	-	-	-	-	-	1
Start	1	$PA_r[I_1..I_{max}]$, $SD_r[J]$	DC	DC	-	-	off	on	-	-	-	clear	2
$Invalid_2$	2	DC	true	DC	-	-	-	-	-	on	-	-	0
Signal_Start	0	$PA_r[I_1] \wedge$ symbol $= PA_r[I_1]$	DC	DC	-	reset enable	on	-	-	-	-	-	1
Start	1	$PA_r[I_1..I_{max}]$, $SD_r[J]$	DC	DC	-	-	off	on	-	-	-	clear	2

transition	S_I	PH_Indication (symbol)	PH_Invalid	MAC_ Reset	T_Neg	TVX	Idle	Rc_ Start	PDU_T k	FO_ Error	TK_ Revd	Flags	S_F
Token	2	$PA_r[I_l..I_{max}]$ $SD_r[JK] \wedge FC_r$ =Token	DC	DC	-	-	-	-	on	-		-	3
Strip_on_Tk	3	$PA_r[I_l..I_{max}]$, $SD_r[JK]$, $FC_r[n,n]$, $PA_r[I_l]$	DC	DC			on	-	-				1
Start	1	$PA_r[I_l..I_{max}]$, $SD_r[J]$	DC	DC	-	-	off	on	-	-	-	clear	2
Token	2	$PA_r[I_l..I_{max}]$ $SD_r[JK] \wedge FC_r$ =Token	DC	DC	-	-	-	-	on	-			3
Format_Error _on_Tk	3	$PA_r[I_l..I_{max}]$, $SD_r[JK]$, $FC_r[n,n]$, $(\neg PA_r[I_l] \vee \neg ED_r[T,T])$	DC	DC	-	-	-	-	-	on			1
Start	1	$PA_r[I_l..I_{max}]$, $SD_r[J]$	DC	DC	-	-	off	on	-	-		clear	2
Token	2	$PA_r[I_l..I_{max}]$ $SD_r[JK] \wedge FC_r$ =Token	DC	DC	-	-	-	-	on	-	-	-	3
Token_Rcvd	3	$PA_r[I_l..I_{max}]$, $SD_r[JK]$, $FC_r[n,n]$, $ED_r[T,T]$	DC	DC	-	-	-	-	-	-	on	-	1
Start	1	$PA_r[I_l..I_{max}]$, $SD_r[J]$	DC	DC	-	-	off	on	-	-	-	clear	2
Token	2	$PA_r[I_l..I_{max}]$ $SD_r[JK] \wedge FC_r$ =Token	DC	DC	-	-	-	-	on	-		-	3
$Invalid_t$	3	DC	true	DC	-	-	-	-	-	on	-	-	0
Signal_Start	0	$PA_r[I_l] \wedge$ symbol $= PA_r[I_l]$	DC	DC	-	reset enable	on	-	-	-	-	-	1
Start	1	$PA_r[I_l..I_{max}]$, $SD_r[J]$	DC	DC	-	-	off	on	-	-	-	clear	2
Token	2	$PA_r[I_l..I_{max}]$ $SD_r[JK] \wedge FC_r$ =Token	DC	DC	-	-	-	-	on	-			3
$Reset_t$	3	DC	DC	true	T_Max	-	-	-	-	-	-	-	0
Signal_Start	0	$PA_r[I_l] \wedge$ symbol $= PA_r[I_l]$	DC	DC	-	reset enable	on	-	-	-	-	-	1

The next iteration of the procedure arbitrarily selects the *signal_start* transition, and the values selected are shown as the second test entered in Table 9. The expected ending state of this second test is 1. At the next iteration, the *invalid*$_t$ transition is chosen, followed by the *signal_start* transition back to *state 1*.

The remaining untested transitions are executed in a similar manner resulting in a final test sequence of 29 steps. The values of the input and output variables for all of these tests are shown in Table 9.

3. Fault Coverage for the FDDI Test Sequence

Thedeterminationoffaultcoverageforthistestsequenceisidenticaltothe TokenBustest sequence. In this specification, Figure 6, there are 4 states and 13 transitions, so there are 52 possible tail states. Subtracting the 13 tail states in a correct IUT, leaves 39 possible type 3 errors. These errors along with the results of each test are listed in Table 10.

TABLE 10: POSSIBLE TYPE 3 ERRORS FOR FIGURE 5

State	Transition	Possible End State	Result
0	$reset_0$	1	**undetected**
0	$reset_0$	2	**undetected**
0	$reset_0$	3	**undetected**
0	*signal_start*	0	detected
0	*signal_start*	2	detected
0	*signal_start*	3	detected
1	$reset_1$	1	detected
1	$reset_1$	2	**undetected**
1	$reset_1$	3	**undetected**
1	$invalid_1$	1	detected
1	$invalid_1$	2	**undetected**
1	$invalid_1$	3	**undetected**
1	*start*	0	detected
1	*start*	1	detected

State	Transition	Possible End State	Result
1	*start*	3	detected
2	$reset_2$	2	detected
2	$reset_2$	1	**undetected**
2	$reset_2$	3	**undetected**
2	$invalid_2$	2	detected
2	$invalid_2$	1	**undetected**
2	$invalid_2$	3	**undetected**
2	*token*	0	detected
2	*token*	1	detected
2	*token*	2	detected
3	$reset_3$	1	**undetected**
3	$reset_3$	2	**undetected**
3	$reset_3$	3	detected
3	$invalid_3$	1	**undetected**
3	$invalid_3$	2	**undetected**
3	$invalid_3$	3	detected
3	*strip_on_tk*	0	detected
3	*strip_on_tk*	2	detected
3	*strip_on_tk*	3	detected
3	*format_error_on_tk*	0	detected
3	*format_error_on_tk*	2	detected
3	*format_error_on_tk*	3	detected
3	*token_rcvd*	0	detected
3	*token_rcvd*	2	detected
3	*token_rcvd*	3	detected

A check of the 39 possible type 3 errors against the test sequence (Table 9) shows that there are 15 faults which are not detected. Note that in each such instance, the error is not detected because it occurs in the presence of one or more converging transitions; any single, type 3 error that does not involve a converging transition will be detected.

For instance, consider the error that could be associated with the *signal_start* transition. In order for this transition to be executed, the machine must be in *state 0* and the predicate $PH\_Indication(symbol) = PAr[I_1] \wedge (symbol = PA_r[I_1])$ must be true. This transition then resets and enables the valid transmission timer (*TVX*) and enables the *idle* signal. If this transition were to end in *state 0*, then either the *signal_start* transition must be executed again, or the $reset_n$ transition is executed. If the *signal_start* transition is executed repeatedly, the machine will appear to be in deadlock, and the error will be detected. The same result, deadlock, will also occur if $reset_n$ is repeatedly executed. Again, as in the previous example, this type of procedure is applied to every potential single, type 3 error to determine if it can be detected.

E. IMPROVING TESTABILITY

1. Token Bus Specification

The fault coverage for the test sequence presented for the Token Bus protocol reveals two ways in which the protocol specification's testability is compromised.

First, from Figure 4, note that *state 1* is a transient state. Since the only transition emanating from this state is *ready*, and the enabling predicate for *ready* is "true," the machine moves from *state 1* back to *state 0* without any input from the tester. The problem is easily resolved, however, by adding the action for the *ready* transition to the action for the *rcv* transition, and then removing the *ready* arc from the predicate action table and the state diagram; this eliminates the need for *state 1* in the state diagram. It is likely that the designer of the specification merely included *state 1* in the original specification, in order to facilitate its understanding. Fortunately, the presence of the transient state in the original

example does not have an adverse effect on the fault coverage provided by the test sequence. However, this is not always the case.

The second problem the test sequence reveals is the existence of the converging transitions mentioned earlier. This situation is more difficult to resolve. The purpose of these two transitions is to bring the machine back to its initial state once it (a) possesses the token but has no data to send (*pass*), or (b) has already sent the maximum number of messages allowed during a single token holding period (*pass-tk*). Since the goal of the transitions is essentially the same (i.e. to pass the token to the next machine) it is difficult for the test sequence to distinguish between them. In this case, the tester should mark the transitions as a potential problem and continue testing.

2. FDDI Specification

The major problem that analysis of the test sequence reveals about this specification is the existence of converging transitions.

In Figure 6 there are thirteen transitions, seven of which are converging transitions. There are four converging *reset* transitions whose purpose is to return the machine to its initial state upon the "resetting" of the MAC layer. The three *invalid* transitions ending in state 0, which indicate to the MAC layer that the physical layer has encountered an invalid frame, are also converging transitions. Although these errors go undetected by the test sequence, the occurrence of any one of them in an implementation that is otherwise error free, would not affect the normal operation of the protocol. This is because all of the *reset* transitions and all of the *invalid* transitions have exactly the same function, respectively.

While it is comforting to note that, in this case, the presence of these errors does not adversely affect the test sequence, this will not always be so. Unfortunately, there is little that the tester or protocol designer can change in this instance without altering the operating characteristics of the machine.

V. PROOF OF FAULT COVERAGE

This chapter is concerned with determining the fault coverage produced by the test method we have been discussing. Essentially, the testing problem is a matter of determining the equivalence of two machines: the specification machine and the machine implementation. If the two machines are equivalent, then the machine implementation, seen as a black box, should generate the same output as the specification machine when both are presented with the same input. Again, since very little can be assumed about the internal structure of the implementation machine, the only way to determine equivalence is to probe both machines with input sequences and compare the resulting output sequences. The problem now is to figure out the right set of inputs and outputs (test sequences) to determine whether or not the machines are equivalent.

In many ways this problem is related to the state verification problem, which has been shown to be unsolvable. However, by limiting the number and type of errors that can occur in an implementation, it is possible to devise a procedure that generates a test sequence which is guaranteed to detect certain types of serious errors. The occurrence of an error in a machine that contains converging transitions presents special problems for the test designer, so care is taken in specifying exactly what constitutes a converging transition.

Two transitions $t_1 = (p_1, a_1)$ and $t_2 = (p_2, a_2)$ are *converging transitions* if all of the following conditions hold:

(1) transitions t_1 and t_2 have different head states but the same tail state;

(2) $(p_1 = p_2)$ or $(p_1 \Rightarrow p_2)$;

(3) $(a_1 = a_2)$ or their actions on all output variables are identical.

In the absence of converging transitions, it is possible to devise a method that will provide good fault coverage, as is evidenced by the following proof. This proof is by contradiction.

Let X' be a protocol implementation under test (IUT) of a protocol machine X, specified by SCM.

Theorem: If (1), X has no converging transitions, and (2), the last transition in the test of X is a UIO sequence, then the test sequence will detect any single type 3 error.

Proof: Suppose not. Then there is such a machine which can be implemented with a single type 3 error, which the test sequence will not detect.

Let H be the state of the IUT at which the error occurs; that is, from which the transition, say 'P', goes to the wrong state (Figure 7).

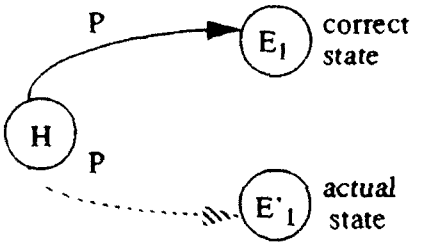

Figure 7 : Type 3 error

Now from the initial state to H, the test sequence has progressed correctly.

TABLE II: EXAMPLE TEST SEQUENCE

S_I	INPUTS	OUTPUTS	S_E
	.	.	
	.	.	
	.	.	
H-1			H
H			E_1

S_I	INPUTS	OUTPUTS	S_E
E_1			E_2
F-1			F
F			F_E

Let {H, E_1, E_2,..., F,...F_E} be the sequence of states as expected to be visited by the test sequence and shown in Table 11.

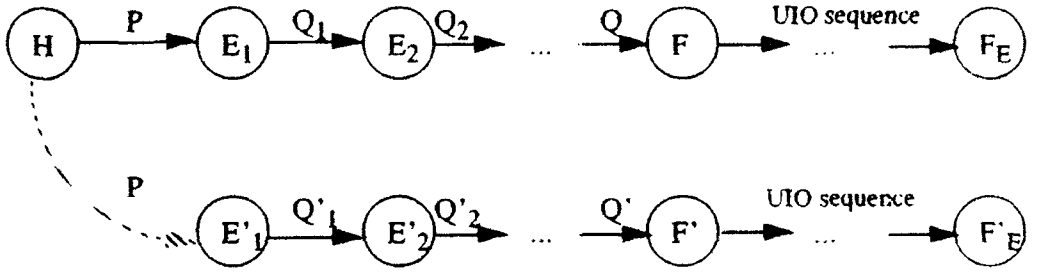

Figure 8 : States Visited in Protocol Machine

Similarly, let E'_1,..., F'_E be the actual sequence taken by the faulty IUT (Figure 8). Since no error is detected, the sequence Q'_1, Q'_2... is exactly equivalent to Q_1, Q_2.... However, F,... F_E is a UIO sequence; hence, F',... F'_E must be exactly the same sequence of states and transitions, or else condition I is violated. Otherwise, F,... F'_E must be F,... F_E. Then there must have been a converging transition in the sequence Q'_1, Q'_2..., which violates condition II. QED.

VI. CONCLUSION

A. CONTRIBUTIONS OF THIS RESEARCH

The goal of this thesis was to present a procedure which generates a test sequence for a communication protocol, that takes as input a protocol specified as a *system of communicating machines*, and gives as output, a complete test sequence. Three recent conformance test procedures were reviewed and their suitability for testing current communication protocols was discussed. A brief specification of two well known local area network protocols was given using SCM and test sequences were generated and analyzed to determine the fault coverage they afforded. Finally, a proof was given that shows the error detection capability of this test method.

The test method introduced here further demonstrates the flexibility of the SCM model. A protocol can be specified, verified and tested using techniques based on this model. In the test procedure, every instance of every transition in the machine specification is tested along with each clause in the enabling predicates. The preliminary steps determine the input and output variables, the sequence generating procedure produces the test sequence, and the refining steps assist in determining fault coverage. It was shown that this method provides good fault coverage in the absence of converging transitions.

The example test sequences for Token Bus and FDDI demonstrate the application of the specification and testing methods associated with the *systems of communicating machines* model. Since these protocols are in wide use today in many networks, their presence as examples illustrates further the usefulness of this test method. Indeed, a test designer would have a difficult time trying to generate a test sequence for these protocols using any of the test methods discussed in Chapter II. Again, using a protocol specification method that has testing in mind yields much better results than using a specification method that was designed without regard to conformance testing.

The proof of fault coverage presented here is important to the test designer because it provides assurance that, under certain circumstances, a serious error in a protocol implementation will be detected. While some of the current literature discusses the correctness of a test sequence, the main emphasis seems to lie in shortening the sequence length. Our procedure, however, emphasizes the ability of the sequence to detect errors rather than achieve an optimal test sequence length. After all, if the protocol test method is automated, the length of the test sequence is of little importance; the fault coverage provided by the sequence is the important part. It is again necessary to emphasize that test methods can only test for the presence of desirable behavior in a protocol machine. It is not possible to exhaustively test for the presence of undesirable behavior since one cannot foresee all possible errors that could occur in an implementation.

B. AREAS FOR FURTHER RESEARCH

Further research might concentrate on extending the error detection capabilities of this method to detect multiple errors or perhaps to detect them in the presence of converging transitions. Since this method treats a protocol implementation as a "black box" the test designer knows nothing about the internal workings of the machine; the tester can only monitor the output of the machine in response to certain inputs. For this reason, UIO sequences are needed to verify the state of the machine at a given instant. It would be interesting to see how different "distinguishing sequences" could be used to better perform this function in the presence of errors.

The recent automation of the specification and analysis portion of the SCM model [ROTH 92], opens the door for the possible automation of the test method introduced here. The procedure is fairly straightforward requiring the intervention of the test designer on matters such as transient states and transitions with multiple clauses, but by starting with simple protocols that do not contain any of these complicating factors it is reasonable to assume that the procedure can be automated.

By showing the types of faults that commonly go undetected by test sequences, this research also provides some insight into designing protocol specifications that are better suited for testing.

REFERENCES

[AHO88] Aho, A. V., Dahbura, A. T., Lee, D., and Uyar, M. U., "An Optimization Technique for Protocol Conformance Test Generation Based on UIO sequences and Rural Chinese Postman Tours," *Proceedings of the 8th Symposium on Protocol Specification, Testing and Verification*, IFIP, June 1988, pp. 75-86.

[HOLT91] Holtzman, Gerard, J., *Design and Validation of Computer Protocols*, Prentice Hall Software Series, Englewood Cliffs, NJ 07974.

[LUND91(a)] Lundy, G. M. and Miller, R. E., "Specification and Analysis of a Data Transfer Protocol Using Systems of Communicating Machines," *Distributed Computing*, Springer-Verlag, December 1991.

[LUND91(b)] Lundy, G. M. and Elmiro, J. L., "A Formal Model of the MAC Layer of an Improved FDDI Protocol," *M. S. Thesis*, Department of Computer Science, Naval Postgraduate School, Monterey, CA, 1991.

[LUND90(a)] Lundy, G.M., and Charbonneau, L. J., "Modeling the Token Bus Protocol with Systems of Communicating Machines", *M. S. Thesis*, Department of Computer Science, Naval Postgraduate School, Monterey, CA, 1990.

[LUND90(b)] Lundy, G. M., and Miller, Raymond E., "Testing Protocol Implementations Based on a Formal Specification," *Proceedings of the 3rd International Workshop on Protocol Test Systems*, IFIP, North Holland, 1990.

[MILL90] Miller, Raymond E., and Paul, Sanjoy, "Two New Approaches to Conformance Testing of Communication Protocols," *TR-90-31*, Department of Computer Science, University of Maryland, 1990.

[ROTH 92] Rothlisberger, M. J., "An Automated Tool for Validation of Network Protocols," *M.S. Thesis*, Department of Computer Science, Naval Postgraduate School, Monterey, CA, September 1992.

[SABN85] Sabnani, K. K., and Dahbura, A. T., "A new technique for generating protocol tests," *Proceedings of the 9th Data Communications Symposium*, IEEE Computer Society Press, September 1985, pp. 36-43.

[SHEN89] Shen, Y. N., Lombardi, F., and Dahbura, A. T., "Protocol Conformance Testing Using Multiple UIO sequences," *Proceedings of the 9th Symposium on Protocol Specification, Testing, and Verification*, IFIP, 1989.

[YANG90] Yang, B., and Ural, J., "Protocol Conformance Test Generation Using Multiple UIO Sequences with Overlapping," *Proceedings of SIGCOMM'90*, Philadelphia, PA, September 1990, pp. 118-125.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 221314

Dudley Knox Library 2
Code 52
Naval Postgraduate School
Monterey, CA 93943

Dr. G. M. Lundy 2
Computer Science Department, Code CSLn
Naval Postgraduate School
Monterey, CA 93943

Director, Force Warfare Aircraft Test Directorate 1
Naval Air Warfare Center, Aircraft Division
Patuxent River, MD 20670

Mr. Michael A. Randall 3
Force Warfare Aircraft Test Directorate
Communications Information Operations Section
Naval Air Warfare Center, Aircraft Division
Patuxent River, MD 20670

Prof. Raymond E. Miller 1
A. V. Williams Bldg.
Dept. of Computer Science
University of Maryland
College Park, MD 20742

Prof. Deepinder Sidhu 1
Dept. of Computer Science
University of Maryland, Baltimore County
Catonsville, MD 21228